1. 함수와 함수형 프로그래밍

1.1 파이썬 함수의 특징

1.2 클로저와 데코레이터

1.3 이터레이터와 제너레이터

1.4 코루틴

```
def add(x,y):
    return x+y

a = add(3,4)
print(a)
```

```
7
```

```
print(type(add))
```

```
<class 'function'>
```

```
print(add)
```

```
<function add at 0x000002081E420798>
```

```
import sys
sys.getrefcount(add)
```

```
2
```

```
add1 = add
a = add1(3,4)
print(a)
```

```
7
```

```
print(type(add1))
```

```
<class 'function'>
```

```
print(add1)
```

```
<function add at 0x000002081E420798>
```

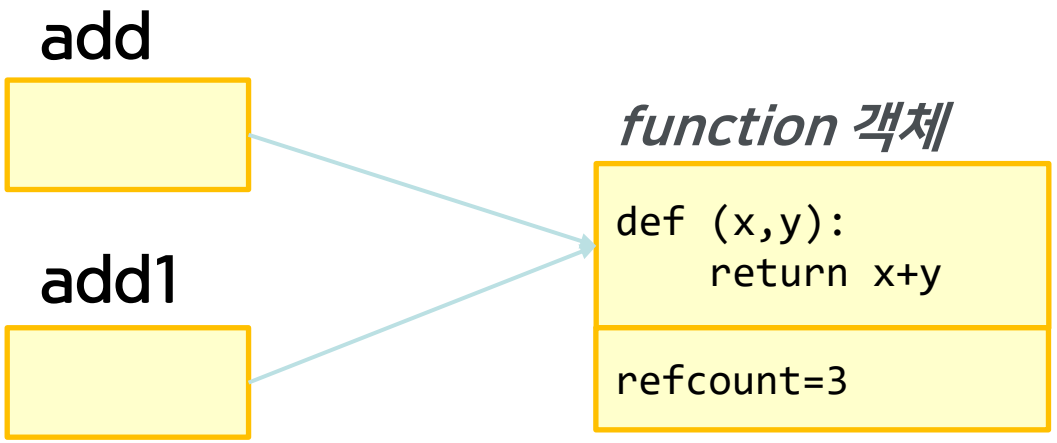
```
sys.getrefcount(add)
```

```
3
```

"파이썬에서는 함수가 객체이다."

```
del add1
sys.getrefcount(add)
```

```
2
```



함수 디폴트 파라메타 주의사항

```
a = 10
def foo(x=a):
    return x
a=5
foo()
```

```
10
```

"함수의 디폴트 파라메타의 값은 함수 정의시에 정해진다."

함수 디폴트 파라메타 주의사항

```
def foo(x, items=[]):
    items.append(x)
    return items

foo(1)
```

```
[1]
```

```
foo(2)
```

```
[1, 2]
```

```
foo(3)
```

```
[1, 2, 3]
```

"디폴트 파라메타가 이전 호출 때의 변경 사항을 담고 있다."

함수 디폴트 파라메타 주의사항

```
def foo(x, items=None):
    if items is None:
        items = []
    items.append(x)
    return items

foo(1)
```

```
[1]
```

```
foo(2)
```

```
[2]
```

```
foo(3)
```

```
[3]
```

"해결책 : 기본 값으로 None을 지정하고 추가적인 검사를 해주는 것이 좋다."

가변 인자

```
def fprintf(file, fmt, *args):
    file.write(fmt % args )

fprintf(sys.stdout, "%d %s %f", 42,
                    "hello world", 3.45)
```

```
42 hello world 3.450000
```

*"인자에 *을 사용하면 남아 있는 모든 인수가 튜플로써 args 변수에 저장 된다."*

```
def printf(fmt, *args):
    fprintf(sys.stdout, fmt, *args)

printf("%d %s %f", 42, "hello world", 3.45)
```

```
42 hello world 3.450000
```

"다른 함수를 호출하고 args를 넘겨준다."

```
def foo(x, w, y, z):
    print(x,w,y,z)

foo(x=3, y=22, w='hello', z=[1,2])
```

```
3 hello 22 [1, 2]
```

```
foo('hello', 3, z=[1,2], y=22)
```

```
hello 3 22 [1, 2]
```

```
foo(3, 22, w='hello', z=[1,2])
```

```
TypeError: foo() got multiple values for argument
'w'
```

"w가 이미 22로 채워져 있는데 다시 'hello'를 할당하려 하면 안된다."

가변 키워드 인수

```
def make_table(data, **parms):
    fgcolor = parms.pop("fgcolor", "black")
    bgcolor = parms.pop("bgcolor", "white")
    width = parms.pop("width", None)
    print(fgcolor)
    print(bgcolor)
    print(width)
    if parms:
        raise TypeError("UnSupported configuration
options %s"% list(parms))

items=[1,2,3]
make_table(items, fgcolor="red", bgcolor="black",
          border=1, borderstyle="grooved",width=400)
```

```
red
black
400
TypeError: UnSupported configuration options
['border', 'borderstyle']
```

키워드 매개변수만 받는 함수

```
def recv(maxsize, *, block=True):
    print(maxsize, block)

recv(8192, block=False)         # Works
try:
    recv(8192, False)           # Fails
except TypeError as e:
    print(e)
```

```
8192 False
recv() takes 1 positional argument but 2 were given
```

*" 방법 1. 이름 없이 *만 사용하면 된다. "*

키워드 매개변수만 받는 함수

```
def minimum(*values, clip=None):
    m = min(values)
    if clip is not None:
        m = clip if clip > m else m
    return m

print(minimum(1, 5, 2, -5, 10))
print(minimum(1, 5, 2, -5, 10, clip=0))
```

```
-5
0
```

*"방법 2. 키워드 매개변수를 *뒤에 넣으면 된다."*

매개변수 전달과 반환 값

```
a = [1, 2, 3, 4, 5]
def square(items):
    for i, x in enumerate(items):
        items[i] = x * x

square(a)
print(a)
```

```
[1, 4, 9, 16, 25]
```

"변경 가능한 객체가 함수에 전달되어 그 값이 변경되면 원래의 객체 값에도 반영된다."

매개변수 전달과 반환 값

```
def divide(a,b):
    return (a//b,a%b)

x, y = divide(10,3)
print(x)
print(y)
```

```
3
1
```

"여러 값을 반환하려면 다음과 같이 튜플에 넣어서 반환하면 된다."

유효 범위 규칙

```
a = 42
def foo():
    a = 13

foo()
print(a)
```

```
42
```

"전역변수 a는 함수 호출 후에도 여전히 42이다. foo함수 내부의 a는 foo함수에 속한 지역변수 이다."

유효 범위 규칙

```
a = 42
b = 37
def foo():
    global a
    a = 13
    b = 0

foo()
print("a=%d, b=%d"%(a,b))
```

```
a=13, b=37
```

"global a는 전역 네임스페이스의 a를 의미한다."

유효 범위 규칙

```
def countdown(start):
    n = start
    def display():
        print('T-minus %d' % n)
    while n > 0 :
        display()
        n -= 1

countdown(3)
```

```
T-minus 3
T-minus 2
T-minus 1
```

"중첩 함수 안의 변수의 이름은 어휘 유효 범위에 따라 찾아 진다."

유효 범위 규칙

```
def countdown(start):
    n = start
    def display():
        print('T-minus %d' % n)
    def decrement():
        n -= 1
    while n > 0 :
        display()
        decrement();

countdown(3)
```

```
T-minus 3
UnboundLocalError: local variable 'n' referenced
before assignment
```

"바깥쪽 함수에서 정의된 지역 변수의 값을 안쪽 함수에서 다시 할당할 수 없다."

유효 범위 규칙

```
def countdown(start):
    n = start
    def display():
        print('T-minus %d' % n)
    def decrement():
        nonlocal n
        n -= 1
    while n > 0 :
        display()
        decrement();

countdown(3)
```

```
T-minus 3
T-minus 2
T-minus 1
```

"바깥쪽 함수에서 정의된 지역 변수의 값을 안쪽 함수에서 다시 할당 하려면 nonlocal을 사용하면 된다. nonlocal은 가장 가까운 바깥쪽 함수에서 정의된 지역 변수를 묶는다."

```
add = lambda x, y : x + y
print(add(3, 5))
```

```
8
```

```
a = [(1, 2), (4, 1), (9, 10),(13, -3)]
a.sort(key=lambda x: x[1])
print(a)
```

```
[(13, -3), (4, 1), (1, 2), (9, 10)]
```

"람다는 1줄 함수 이다. 다른 프로그래밍 언어에서는 익명 함수 라고도 부른다.

프로그램 내에서 함수를 재사용 하지 않으려는 경우 lambda 함수를 쓸 수 있다.

일반함수와 형태도 비슷하고 비슷하게 작동한다."

```
x = 10
a = lambda y : x + y
x = 20
b = lambda y : x + y

print(a(10))
print(b(10))
```

```
30
30
```

"람다에서 사용한 x값이 실행 시간에 달라지는 변수이다."

```
x = 15
print(a(10))
x = 3
print(a(10))
```

```
25
13
```

```
x = 10
a = lambda y, x=x : x + y
x = 20
b = lambda y, x=x : x + y

print(a(10))
print(b(10))
```

```
20
30
```

"함수를 정의할 때 특정 값을 고정하고 싶으면 그 값을 디폴트 파라메타로 지정하면 된다."

```
funcs = [lambda x: x+n for n in range(5)]
for f in funcs:
    print(f(0), end=' ')
```

```
4 4 4 4 4
```

"n 값이 변할 것이라고 기대 하지만 람다 함수는 가장 마지막 값을 사용한다."

```
funcs = [lambda x, n=n: x+n for n in range(5)]
for f in funcs:
    print(f(0), end=' ')
```

```
0 1 2 3 4
```

"n 값을 함수를 정의하는 시점의 값으로 고정해 놓고 사용한다."

함수 인자에 메타데이터 넣기

```
def add(x:int, y:int) -> int:
    return x + y

help(add)
```

```
Help on function add in module __main__:
add(x: int, y: int) -> int
```

"인자에 정보를 추가해서 다른 사람이 함수를 어떻게 사용하는지 알 수 있도록 할 수 있다."

```
add.__annotations__
```

```
{'x': int, 'y': int, 'return': int}
```

"함수의 주석은 함수의 __annotations__에 저장된다."

1. 함수와 함수형 프로그래밍

1.1 파이썬 함수의 특징

1.2 클로저와 데코레이터

1.3 이터레이터와 제너레이터

1.4 코루틴

```
def calc():
    a = 3
    b = 5
    def mul_add(x):
        return a * x + b
    return mul_add

c = calc()
print(c(1), c(2), c(3), c(4), c(5))
```

```
8 11 14 17 20
```

"함수가 데이터로 취급될 때는 함수가 정의된 곳의 주변 환경 정보가 함께 저장 된다.

함수를 구성하는 문장과 실행 환경을 함께 묶은 것을 클로저 라고 부른다."

```
c.__closure__
```

```
(<cell at 0x000001B0D2CA64C8: int object at
0x00007FFF8307A1D0>,
 <cell at 0x000001B0D2CA6378: int object at
0x00007FFF8307A210>)
```

```
c.__closure__[0].cell_contents
```

```
3
```

```
c.__closure__[1].cell_contents
```

```
5
```

"클로저 내부에 __closure__에 free variable이 저장되어 있다."

```
def calc():
    a = 3
    b = 5
    return lambda x: a * x + b

c = calc()
print(c(1), c(2), c(3), c(4), c(5))
```

```
8 11 14 17 20
```

"구현이 간단한 수식인 경우 람다를 이용하여 클로저로 사용할 수 있다.

내부 함수를 직접 호출할 일이 없으므로 이름이 없어도 된다."

클로저 내부에서 정의한 변수에 접근

```
def sample():
    n = 0
    def func():
        print('n =', n)

    def get_n():
        return n

    def set_n(value):
        nonlocal n
        n = value

    func.get_n = get_n
    func.set_n = set_n
    return func
```

```
if __name__ == '__main__':
    f = sample()
    f()
    n = 0
    f.set_n(10)
    f()
    print(f.get_n())
```

```
n = 0
n = 10
10
```

"주요한 두가지 기능

첫번째 : nonlocal 선언으로 내부 변수를 수정하는 함수를 작성
두번째 : 접근 메소드를 클로저 함수에 붙여 마치 인스턴스 메소드인 것처럼 동작하는 것"

```
def big_number(n):
    return n ** n ** n

def make_func_alarm(func):
    def new_func(*args, **kwargs):
        print("함수를 시작합니다.")
        result = func(*args, **kwargs)
        print('함수를 종료합니다.')
        return result
    return new_func

new_func = make_func_alarm(big_number)
new_func(6)
```

```
함수를 시작합니다.
함수를 종료합니다.
```

"함수를 인자를 받아서, 함수 전후에 새로운 기능을 추가한 함수를 만들 수도 있다."

```
import time

def make_time_checker(func):
      def new_func(*args, **kwargs):
            start_time = time.perf_counter()
            result = func(*args, **kwargs)
            end_time = time.perf_counter()
            print('실행시간:', end_time - start_time)
            return result
      return new_func

new_func = make_time_checker(big_number)
new_func(7)
```

```
실행시간: 0.14249400000016976
```

"함수가 실행되는 시간을 측정하는 기능을 가지는 새로운 함수를 만들어서 리턴할 수도 있다."

```
import time

def make_time_checker(func):
    def new_func(*args, **kwargs):
        start_time = time.perf_counter()
        result = func(*args, **kwargs)
        end_time = time.perf_counter()
        print('실행시간:', end_time - start_time)
        return result
    return new_func

@make_time_checker
def big_number(n):
      return n ** n ** n

big_number(7)
```

```
실행시간: 0.14843939999991562
```

"@make_time_checker 문법은 데코레이터 이다."

```
enable_tracing = True
if enable_tracing:
    debug_log = open("debug.log","w")

def trace(func):
    if enable_tracing:
        def callf(*args,**kwargs):
            debug_log.write("Calling %s: %s, %s\n"%
                            (func.__name__,args,kwargs))
            r = func(*args,**kwargs)
            debug_log.write("%s returned %d\n"%
                            (func.__name__,r))
            return r
        return callf
    else :
        return func
```

```
@trace
def square(x):
    return x*x

square(100)
```

```
10000
```

debug.log 파일 내용

```
Calling square: (100,), {}
square returned 10000
```

1. 함수와 함수형 프로그래밍

1.1 파이썬 함수의 특징

1.2 클로저와 데코레이터

1.3 이터레이터와 제너레이터

1.4 코루틴

```
with open("passwd") as f:
    try:
        while True:
            line = next(f)
            print(line, end='')
    except StopIteration:
        pass
```

```
#root:x:0:0:root:/root:/bin/bash
#daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
#bin:x:2:2:bin:/bin:/usr/sbin/nologin
#sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
...
```

"문제:순환 가능한 아이템에 접근할 때 for 순환문을 사용하고 싶지 않다.
해결 : 이터레이터를 사용하면 된다."

수동으로 이터레이터 소비

```
items = [1, 2, 3]
it = iter(items)
next(it)
```

```
1
```

```
next(it)
```

```
2
```

```
next(it)
```

```
3
```

```
next(it)
```

```
---------------------------------------------------------
StopIteration                                 Traceback
(most recent call last)
<ipython-input-33-bc1ab118995a> in <module>
----> 1 next(it)
StopIteration:
```

```
class Node:
    def __init__(self, value):
        self._value = value
        self._children = []

    def __repr__(self):
        return 'Node({!r})'.format(self._value)

    def add_child(self, node):
        self._children.append(node)

    def __iter__(self):
        return iter(self._children)
```

```
if __name__ == '__main__':
    root = Node(0)
    child1 = Node(1)
    child2 = Node(2)
    root.add_child(child1)
    root.add_child(child2)
    for ch in root:
        print(ch)
```

```
Node(1)
Node(2)
```

```
def frange(start, stop, increment):
    x = start
    while x < stop:
        yield x
        x += increment

for n in frange(0, 4, 0.5):
    print(n , end=' ')
```

```
0 0.5 1.0 1.5 2.0 2.5 3.0 3.5
```

```
list(frange(0, 1, 0.125))
```

```
[0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875]
```

제너레이터로 순환 패턴 생성

```
def countdown(n):
    print('Starting to count from',n)
    while n > 0:
        yield n
        n -= 1
    print('Done!')

c = countdown(3)
c
```

```
<generator object countdown at 0x000001B0D2D36F48>
```

```
next(c)
```

```
Starting to count from 3
3
```

```
next(c)
```

```
2
```

```
next(c)
```

```
1
```

```
next(c)
```

```
Done!
---------------------------------------------------------------
StopIteration                                 Traceback
(most recent call last)
<ipython-input-42-e846efec376d> in <module>
----> 1 next(c)

StopIteration:
```

이터레이터 프로토콜 구현

```
class Node:
    def __init__(self, value):
        self._value = value
        self._children = []

    def __repr__(self):
        return 'Node({!r})'.format(self._value)

    def add_child(self, node):
        self._children.append(node)

    def __iter__(self):
        return iter(self._children)

    def depth_first(self):
        yield self
        for c in self:
            yield from c.depth_first()
```

```
if __name__ == '__main__':
    root = Node(0)
    child1 = Node(1)
    child2 = Node(2)
    root.add_child(child1)
    root.add_child(child2)
    child1.add_child(Node(3))
    child1.add_child(Node(4))
    child2.add_child(Node(5))

    for ch in root.depth_first():
        print(ch, end=' ')
```

```
Node(0) Node(1) Node(3) Node(4) Node(2) Node(5)
```

역방향 순환

```
a = [1,2,3,4]
for x in reversed(a):
    print(x, end=' ')
```

```
4 3 2 1
```

```
class Countdown:
    def __init__(self, start):
        self.start = start

    def __iter__(self):
        n = self.start
        while n > 0:
            yield n
            n -= 1

    def __reversed__(self):
        n = 1
        while n <= self.start:
            yield n
            n += 1
```

```
c = Countdown(5)
print("Forward:")
for x in c:
    print(x, end=' ')
print('\n')
print("Reverse:")
for x in reversed(c):
    print(x, end=' ')
```

```
Forward:
5 4 3 2 1

Reverse:
1 2 3 4 5
```

추가 상태를 가진 제너레이터 함수 정의

```
from collections import deque

class linehistory:
    def __init__(self, lines, histlen=3):
        self.lines = lines
        self.history = deque(maxlen=histlen)

    def __iter__(self):
        for lineno, line in enumerate(self.lines,1):
            self.history.append((lineno, line))
            yield line

    def clear(self):
        self.history.clear()
```

추가 상태를 가진 제너레이터 함수 정의

```
with open('somefile.txt') as f:
     lines = linehistory(f)
     for line in lines:
         if 'python' in line:
             for lineno, hline in lines.history:
                 print('{}:{}'.format(lineno, hline), end='')
```

```
2:this is a test
3:of iterating over lines with a history
4:python is fun
```

이터레이터의 일부 얻기

```
def count(n):
    while True:
        yield n
        n += 1

c = count(0)
c[5:10]
```

```
---------------------------------------------------------------
TypeError                                 Traceback
(most recent call last)
<ipython-input-75-0e5ab4786dd3> in <module>
      5
      6 c = count(0)
----> 7 c[5:10]

TypeError: 'generator' object is not subscriptable
```

이터레이터의 일부 얻기

```
import itertools

for x in itertools.islice(c, 10, 20):
    print(x)
```

```
10
11
12
13
14
15
16
17
18
19
```

순환 객체 첫 번째 부분 건너뛰기

```
with open("passwd") as f:
    for line in f:
        print(line, end='')
```

```
#root:x:0:0:root:/root:/bin/bash
#daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
#bin:x:2:2:bin:/bin:/usr/sbin/nologin
#sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
...
```

순환 객체 첫 번째 부분 건너뛰기

```
from itertools import dropwhile
with open("passwd") as f:
    for line in dropwhile(lambda line: line.startswith('#'), f):
        print(line, end='')
```

```
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
...
```

순환 객체 첫 번째 부분 건너뛰기

```
from itertools import islice
items = ['a', 'b', 'c', 1, 4, 10, 15]
for x in islice(items, 3, None):
    print(x, end=' ')
```

```
1 4 10 15
```

```
with open('passwd') as f:
    while True:
        line = next(f)
        if not line.startswith('#'):
            break
    while line:
        print(line, end='')
        line = next(f, None)
```

```
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
...
```

인덱스-값 페어 시퀀스 순환

```
my_list = ['a', 'b', 'c']
for idx, val in enumerate(my_list):
    print(idx, val)
```

```
0 a
1 b
2 c
```

```
my_list = ['a', 'b', 'c']
for idx, val in enumerate(my_list,1):
    print(idx, val)
```

```
1 a
2 b
3 c
```

인덱스-값 페어 시퀀스 순환

```
idx=1
for val in my_list:
    print(idx, val)
    idx+=1
```

```
1 a
2 b
3 c
```

여러 시퀀스 동시에 순환

```
xpts = [1, 5, 4, 2, 10, 7]
ypts = [101, 78, 37, 15, 62, 99]
for x, y in zip(xpts,ypts):
    print(x, y)
```

```
1 101
5 78
4 37
2 15
10 62
7 99
```

여러 시퀀스 동시에 순환

```
a = [1, 2, 3]
b = ['w', 'x', 'y', 'z']
for i in zip(a,b):
    print(i)
```

```
(1, 'w')
(2, 'x')
(3, 'y')
```

```
from itertools import zip_longest
for i in zip_longest(a,b):
    print(i)
```

```
(1, 'w')
(2, 'x')
(3, 'y')
(None, 'z')
```

1. 함수와 함수형 프로그래밍

1.1 파이썬 함수의 특징

1.2 클로저와 데코레이터

1.3 이터레이터와 제너레이터

1.4 코루틴

코루틴과 yield 표현식

```
def receiver():
    print("Ready to receive")
    while True:
        n = yield
        print("Got %s" % n)

r = receiver()
next(r)
```

```
Ready to receive
```

```
r.send(1)
```

```
Got 1
```

```
r.send(2)
```

```
Got 2
```

```
r.send("Hello")
```

```
Got Hello
```

```
def coroutine(func):
    def start(*args, **kwargs):
        g = func(*args, **kwargs)
        next(g)
        return g
    return start

@coroutine
def receiver():
    print("Ready to receive")
    while True:
        n = (yield)
        print("Got %s" % n)
```

```
r = receiver()
r.send("Hello World")
```

```
Ready to receive
Got Hello World
```

```
r.close()
r.send(4)
```

```
---------------------------------------------------------------
StopIteration                             Traceback
(most recent call last)
<ipython-input-104-a529e2e7c9bc> in <module>
      1 r.close()
----> 2 r.send(4)

StopIteration:
```

```
import os
import fnmatch
def find_files(topdir, pattern):
    for path, dirname, filelist in os.walk(topdir):
        for name in filelist:
            if fnmatch.fnmatch(name, pattern):
                yield os.path.join(path,name)

def opener(filenames):
    for name in filenames:
        f = open(name)
        yield f

def cat(filelist):
    for f in filelist:
        for line in f:
            yield line

def grep(pattern, filelist):
    for line in lines:
        if pattern in line:
            yield line
```

```
passwd = find_files(".", "passwd")
files = opener(passwd)
lines = cat(files)
pylines = grep("linux", lines)
for line in pylines:
    print(line)
```

```
linux:x:1000:1000:linux,,,:/home/linux:/bin/bash
```

```
import os
import fnmatch

@coroutine
def find_files(target):
    while True:
        topdir, pattern = (yield)
        for path, dirname, filelist in os.walk(topdir):
            for name in filelist:
                if fnmatch.fnmatch(name, pattern):
                    target.send(os.path.join(path,name))

@coroutine
def opener(target):
    while True:
        name = (yield)
        f = open(name)
        target.send(f)
```

```
@coroutine
def cat(target):
    while True:
        f = (yield)
        for line in f:
            target.send(line)

@coroutine
def grep(pattern, target):
    while True:
        line = (yield)
        if pattern in line:
            target.send(line)

@coroutine
def printer():
    while True:
        line = (yield)
        print(line)
```

```
finder = find_files(opener(cat(grep("linux", printer()))))
finder.send((".", "passwd"))
```

```
linux:x:1000:1000:linux,,,:/home/linux:/bin/bash
```

2. 클래스와 객체

2.1 클래스와 객체 I

2.2 클래스와 객체 II

2.3 클래스와 객체 III

인스턴스의 문자열 표현식 변형

```
class Pair:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __repr__(self):
        return 'Pair({0.x!r}, {0.y!r})'.format(self)
    def __str__(self):
        return '({0.x}, {0.y})'.format(self)

p = Pair(3, 4)
p
```

```
Pair(3, 4)
```

"인스턴스의 문자열 표현식을 바꾸려면 __str__()와 __repr__() 메소드를 정의 한다."

인스턴스의 문자열 표현식 변형

```
print(p)
```

```
(3, 4)
```

"객체를 출력할 경우 기본적으로는 __str__()함수가 호출 된다."

```
print('p is {0!r}'.format(p))
```

```
p is Pair(3, 4)
```

"!r은 기본 값으로 __repr__()함수가 호출 된다."

```
print('p is {0}'.format(p))
```

```
p is (3, 4)
```

"{}은 기본 값으로 __str__()함수가 호출 된다."

"__repr__()는 eval(repr(x)) == x와 같은 텍스트를 만드는 것이 표준이다."

```
from socket import socket, AF_INET, SOCK_STREAM

class LazyConnection:
    def __init__(self, address, family=AF_INET, type=SOCK_STREAM):
        self.address = address
        self.family = AF_INET
        self.type = SOCK_STREAM
        self.sock = None

    def __enter__(self):
        if self.sock is not None:
            raise RuntimeError('Already connected')
        self.sock = socket(self.family, self.type)
        self.sock.connect(self.address)
        return self.sock

    def __exit__(self, exc_ty, exc_val, tb):
        self.sock.close()
        self.sock = None
```

객체의 콘텍스트 관리 프로토콜 지원

```
if __name__ == '__main__':
    from functools import partial

    c = LazyConnection(('www.python.org', 80))
    with c as s:
        s.send(b'GET /index.html HTTP/1.0\r\n')
        s.send(b'Host: www.python.org\r\n')
        s.send(b'\r\n')
        resp = b''.join(iter(partial(s.recv, 8192), b''))

    print('Got %d bytes' % len(resp))
```

```
Got 392 bytes
```

"객체가 콘텍스트 관리 프로토콜(with 구문)을 지원하게 구현"

with문을 만나면 __enter__() 메소드가 호출된다.
with문을 빠져 나갈때는 __exit__() 메소드가 호출된다.

인스턴스를 많이 생성할 때 메모리 절약

```
class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

d1 = Date(2020, 6, 1)
d1.__dict__
```

```
{'year': 2020, 'month': 6, 'day': 1}
```

"객체 인스턴스 마다 내부에 멤버를 저장할 목적으로 딕셔너리를 구성한다. 이는 메모리 낭비의 원인이 된다."

인스턴스를 많이 생성할 때 메모리 절약

```
class Date:
    __slots__ = ['year', 'month', 'day']
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

d1 = Date(2020, 6, 1)
dir(d1)
```

```
['__class__',
 '__delattr__',
 '__dir__',
 '__doc__',
 ...
```

"__slots__을 정의하면 파이썬은 인스턴스에 훨씬 더 압축된 내부 표현 식을 사용한다.
인트턴스 마다 딕셔너리를 구성하지 않고 튜플이나 리스트 값은 부피가 작은 고정 배열로 인스턴스가 만들어 진다. "

관리 속성 만들기

```
class Person:
    def __init__(self, first_name):
        self.first_name = first_name

    @property
    def first_name(self):
        return self._first_name

    @first_name.setter
    def first_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._first_name = value
```

"인스턴스 속성을 얻거나 설정할 때 추가적인 처리(타입 체크, 검증 등)를 하고 싶다."

```
if __name__ == '__main__':
    a = Person('Guido')
    print(a.first_name)
    a.first_name = 'Dave'
    print(a.first_name)
    try:
        a.first_name = 42
    except TypeError as e:
        print(e)
```

```
Guido
Dave
Expected a string
```

"@first_name.setter로 데코레이터를 지정하면 해당 속성을 변경하려 할 때 자동으로 호출 된다.

이때 타입을 검사할 수 있고 문자열이 아닌경우 예외를 던진다."